Protecting shared state in a multi-core world: a comparison between traditional locking and transactional memory
Contents

1 Introduction 3
  1.1 Preface ........................................ 3
  1.2 The Free Lunch is over ...................... 3
  1.3 GX WebManager ............................... 4
  1.4 Research question ............................. 4
    1.4.1 What is scalability ....................... 4

I Introduction into concurrent programming in Java 6

2 Concurrent programming in Java 7
  2.1 The problem with concurrent programming: Unprotected shared state .......... 7
    2.1.1 Generic example ......................... 8
  2.2 Protecting shared state access .................. 8
    2.2.1 Synchronized ............................... 9
    2.2.2 ReadWriteLock ................................ 10
    2.2.3 Transactional Memory ....................... 11

II Case analysis: GX WebManager 13

3 Architecture of GX WebManager 15

4 Analysis of concurrent properties of GX WebManager 18
  4.1 The Cache component .......................... 18
    4.1.1 URLMap .................................... 19
    4.1.2 URL Manager .............................. 19

5 Measuring concurrent performance characteristics of the cache component 21
  5.1 Test setup .................................... 21
  5.2 Hypothesis of test results .................... 22
    5.2.1 Results .................................. 22
    5.2.2 Analysis of results ....................... 23
    5.2.3 Explaining the results .................... 24
    5.2.4 Conclusion ................................ 26

III Comparison of ReadWriteLock vs Software Transactional Memory 27

6 Introduction 28
  6.1 Research question ............................. 28
  6.2 TestSetup ..................................... 29
6.3 Test parameters ................................................................. 29

7 Test results & analysis ......................................................... 31
  7.1 ReadWriteLock .............................................................. 31
    7.1.1 Response times results ............................................ 31
    7.1.2 CPU Utilization results .......................................... 34
    7.1.3 Throughput Results ............................................... 36
    7.1.4 ReadWriteLock Conclusion ........................................ 37
  7.2 Deuce ................................ ........................................ 39
    7.2.1 Response times Results ........................................... 39
    7.2.2 CPU Utilization Results ......................................... 41
    7.2.3 Throughput Results ............................................... 43
    7.2.4 Transaction results ............................................... 45
    7.2.5 Deuce Conclusion .................................................. 46
  7.3 ReadWriteLock vs Deuce ............................................... 48
    7.3.1 0% writes ............................................................ 48
    7.3.2 10% and 20% writes .............................................. 50
    7.3.3 Conclusions ........................................................ 53
Chapter 1

Introduction

1.1 Preface

In my early years as a software engineer, concurrency already intrigued me: how is it possible to execute the same blocks of code in parallel and share the same data? Although I do understand the principles of concurrency it still keeps intriguing me, but now from a different perspective: how should concurrency be used to improve the user’s experience and quality of a software component?

Within this master thesis I will show, analyze and discuss the necessity and issues caused by concurrency including advantages and disadvantages of possible solutions. In Chapter 1 a short introduction is given into concurrent programming. What exactly is the problem and what solutions are available? GX WebManager will act as a use case and it’s concurrent characteristics will be investigated and measured. In the last chapter two completely different approaches to issues caused by concurrency are compared to each other under varying circumstances.

The following sections will draw a complete context of this thesis and will formulate a research question that will be answered within the upcoming three parts of this thesis.

1.2 The Free Lunch is over

The processors we work with today have come a long way and have become technical and complex objects to directly work with. Techniques such as branch prediction, instruction pipelining, more complex instructions, memory caches and hyperthreading push processor performance to the limit. Also hardware developments made it possible to fit more transistors within a single processor (also known as Moore’s Law) or reduce the size of a processor-die to reduce heat generation. Another performance improvement is the increase of the clock frequency. A frequency that is two times as high performs the same amount of work in half of the time. This phenomena which is also known as frequency scaling dominated the performance characteristics of processors.

From the programmers point of view this development is positive. Running a program on a processor that has a higher frequency automatically reduces execution time as more commands are processed during execution. The concept of improved performance by using a more powerful processor is also known as ’Free Lunch’ and was the trend up to year 2003. The following years up to today show a different trend: almost no frequency scaling and the introduction of the multi-core concept.

In March 2005 Herb Sutter wrote an article that stated that the so called Free Lunch was over[20] and that software design should make a fundamental turn to concurrency to squeeze the last drop out of the multi-core processors. Whereas concurrency was seen as something that could break your program it has become a fundamental part of any modern software design. Single threaded is dead: the future is concurrent
1.3 GX WebManager

GX WebManager is a Web Content Management System (WCMS) developed by GX Software that supports companies and organizations using the internet as a strategic channel. In its essence, GX WebManager allows editors to organize the content displayed via a website. One important aspect of GX WebManager is the performance characteristics as experienced by the end-user: the contents should be displayed without too much delay, even when the load substantially increases. To achieve this target, GX Software has designed the architecture of GX WebManager such that the latency of a requests will be as low as possible. Changes to for example news items should also be available instantly while preserving fast response times.

1.4 Research question

The task of delivering low response times when the load on a website increases is difficult. A complete cycle from a HTTP request to the contents of the requested page on disk or in memory and back will pass through many system layers and components. If one of these components has less capacity than the previous ones, a bottleneck will occur. These bottlenecks can appear in components such as the network bandwidth, I/O access, algorithm being used or processor capacity. The trend within CPU development has moved from frequency scaling to a multi-core solution. This paradigm shift introduces a challenge for programmers, compilers and language designers to write applications that can deliver the ever growing demands of its users.

Software designers and engineers face the task to design and build an application that maximizes the effective use of available resources (including CPU resources). A single-threaded application’s ability to cope with an increasing load is limited to increasing the CPU frequency and should actually be avoided. A multi-threaded application can deal better with increasing load as the multiple threads can better utilize multiple processor cores. However, the utilization rate is highly dependent on the amount of code that can be executed in parallel. Amdahl [6] and later Gustafon [10] showed that even a small section that cannot be parallelized reduces the total speedup significantly. Besides speedup, scalability is very important and can only be realized within a multi-thread application. Also the scalability is affected by the amount of code that can be executed in parallel, but in a slightly different way. Speedup is dependent on the amount of parallelizable sections, scalability develops with a (theoretical) infinite amount of concurrent threads running the same piece of code in parallel.

With the shift from frequency scaling to a multi-core paradigm (see chapter 1.2 on the preceding page) in mind; how can GX WebManager benefit from these extra cores? Within the multi-core paradigm, benefitting from these extras does not necessarily reduce the response times (see Amdahl [6] and Gustafon [10] but should provide better scalability. Exaggerated: a single core for each visitor. This context is different from the frequency scaling paradigm where average response times should drop with faster processors and as a side-effect the scalability also increases.

1.4.1 What is scalability

Scalability is a term indicating an application’s capability to increase its output when the input increases. The process of transforming an individual input to output takes an amount of time. This duration influences the amount of work (transformation of input) that can be processed within a fixed time period. An application scales properly if increases in load has limited (or ideally zero) effect on the amount of time needed to process a single unit of work. In an ideal situation, doubling the load results into a doubled output and is considered to deliver linear scalability.

Bottlenecks within the application cause the time to process a single unit of work to rise. The greater this increase the lesser units of work can be completed within the fixed time period, hence the scalability becomes near-linear. In some situations the bottlenecks can become so severe that absolutely no scalability is achieved.
For GX WebManager scalability is important to guarantee low response times (which represent a single unit of work) while the load increases. Upon a certain point the amount of processors will create a bottleneck as result of the following two causes. The first cause is complete utilization of the CPU’s computational resources. This effect is visible as 100% load per processor core within the operating system. Another cause is that the increasing amount of software threads cannot be directly mapped to the limited amount of processor threads that can run concurrently. Depending on the context, higher frequency, more core or a combination of these two will resolve this bottleneck and move it to another part of the whole system.
Part I

Introduction into concurrent programming in Java: trends, problems & solutions
Chapter 2

Concurrent programming in Java

Although processors can only work on one instruction at the time, programs running on these processors often contain many commands that need to run in parallel. The very basic concept of receiving input from the user, processing the data and displaying the results are three distinct processes that need to run separate from each other. Viewing a web-page or listening to music is only possible when multiple tasks are run at the same time with each having its own responsibilities. Concurrency is found in all levels of the OSI model, including the higher layers.

Parallelism in Java is based upon the concept of threads. From the programmers point of view a java application always has at least one thread, which is also called the main thread. Besides this main thread, the Java Virtual Machine also has threads running that perform system tasks such as garbage collection. All these threads (both system and application) run within the same process. [4]

2.1 The problem with concurrent programming: Unprotected shared state

Java is an imperative and object-oriented programming language: execution of a java application will change states of a set of class instances (also know as objects) until some final state is reached. The state of an object is determined by the values of its member variables (also know as fields). When a field value changes, the state of the object it belongs to also changes.

The same applies within a concurrent environment where multiple threads share the set of objects that makes up an application. This phenomena is also known as shared state and is caused by a combination of the fundamental concept of imperative programming and concurrency. When designing and writing an application, architects and programmers need to be aware of this behavior and prevent shared state or protect the fields of an object such that it cannot enter an illegal state.

It is important to understand that shared state is not a problem but a manner to share data between different threads. However, with improper use it can become a problem and therefore it needs to handled carefully. Shared state becomes a problem when the action of changing shared state becomes dividable and that the information carried with it becomes inconsistent.

Unexpected or faulty behavior of an application as a result of shared state is not the only problem. More shared state within an application results into more protection mechanisms. Depending on the characteristics of the program (how many write accesses?) and protection mechanisms the overall speedup as result of more CPU resources will be lower. The same applies for scalability of throughput.

Critical section Although it is not totally uncommon, shared state can be compositional and is often accessed multiple times within a block of code. Critical section is a term that identifies
a block of code that accesses a group of shared states and when not protected can result into unexpected behavior.

2.1.1 Generic example

A common use of shared state is to use a field to determine which actions needs to be executed. After checking the value of such a field, another thread modifies the field and the action that is executing fails due to the changed state. State is not always changed explicitly by the programmer, but can sometimes occur as a result of another action. Such a change is also known as a side-effect of function execution. This makes tracing the root cause problematic as the exact circumstances at time of change need to be known. Side-effects are quite common in an java application, but in combination with a race condition can create very problematic situations that are difficult to solve. The problem sketched above is written in pseudo-code below and will be used to give an example of a race condition.

Listing 2.1: Generic example of shared state problem

```
public class Example{
    private object sharedObject

    public void doSomething(){
        if(sharedObject == null)
            createSharedObject();
        else
            performActionOnSharedObject()
    }
}
```

A race-condition within the pseudo-code above could look as follows:

1. Thread 1 evaluates sharedObject == null to false and is going to execute performActionOnSharedObject
2. Thread 2 nullifies sharedObject
3. Thread 1 expects sharedObject to be a object reference but finds none and throws a Null-Pointer exception.

2.2 Protecting shared state access

Perhaps trivial but still effective enough to be mentioned: the best solution for the problem is to remove all shared state and make the critical section stateless. Another solution is to not protect the critical sections with some mechanism, but to confine threads accessing these to a separate thread that is responsible for managing access. Examples of these concepts can be found within Swing's event dispatch model and the Java DataBase Connectivity (JDBC) Connection object where each instance is confined to a single separate thread.

In 1974 Edgar Dijkstra wrote the first paper about semaphores ("Over seinpalen" [8]) that proposes two types of semaphore (binary & counting) as a solution for the problem caused by unprotected shared state. With a semaphore the access to shared state is protected and processes were synchronized. All threads accessing the shared state should pass through and possibly wait for the semaphore. Depending on the type of semaphore a thread could gain access and modify the shared state. This concept is also known as locking. In Java a lock is obtained on a monitor and this monitor is associated with an object. Synchronization is an implementation of a binary semaphore and from a semantic point of view the perfect solution for protecting shared state as it is simple to understand and implement. In the earlier versions of Java up to 1.5.0 , synchronization was the
standard and only solution for shared state. Although effective, the synchronized mechanism has some big shortcomings. Not only the scalability was completely ruined it also lacked expressivity needed for complex problems. Better and more flexible solutions were developed. These solutions can be divided into two categories:

**Blocking** When threads can be blocked when trying to access a protected shared state, then the protection mechanism is considered to be blocking. As mentioned above, *Synchronized* is an implementation of a binary semaphore and allows only one thread at the time to access and modify shared state. Another solution is called *ReadWriteLock* and combines a binary semaphore (for write access) and a counting semaphore (for read access) for more fine-grained locking and shows better performance. Although in a read-only situation the mechanism seems to be non-blocking as all threads can access the shared state (counter is infinite), a write-access will block all access (writers & readers).

**Non-blocking** A completely different solution is the concept of (Software) Transactional Memory where access to a critical section is managed within a transactional system. These transactions run concurrently and when they are committed the current state is compared with the initial state; if they differ the transaction is rollbacked and retried (also see 2.2.3 on page 11).

To decide which solution is the best for the given problem it is important to understand the underlying concept and to know how it should be used. Usability of a shared state protection mechanism is very important; difficult solutions will lead to programs that are difficult to understand and test, resulting in a higher chance of errors. Another important aspect of these solutions is the scalability, especially with the current development in processor manufacturing (see chapter 1.2 on page 3). The next paragraphs will discuss the concepts and usability of several locking mechanisms available.

### 2.2.1 Synchronized

As explained in the first paragraph of this section, synchronization is a binary semaphore and locks a monitor that prevents concurrent access to the protected critical section. The has 2 important consequences, generally referred to as atomicity and visibility. Atomicity ensures that only one thread is running and other concurrent threads will be blocked. The second consequence, visibility ensures that the memory is kept in a consistent state and that all other threads receive the latest values despite local thread caches, compiler optimizations, registers and processor caches. According to the Java Memory Model, exiting a synchronization block automatically establishes a happens-before relationship with any subsequent invocation of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads [4].

Within the Java language, the synchronized keyword can be used to synchronize methods or statements. The latter is more fine-grained and can also synchronize on user-defined monitors. Synchronization at method level uses the class instance (this) as monitor. This behavior has the disadvantage that methods accessing unrelated shared state will block each other. When synchronized is used in combination with a group of statements it is possible to specify an object that acts as a monitor. Using this technique, a more fine-grained solution can be designed. Using custom objects as locking has the disadvantage that the programmer must be aware which monitors should be used for specific sections. The same applies for synchronized methods when such a method modifies a public field of another class. Note that both types of synchronization can be swapped without too much effort: a group of statements that is wrapped within a synchronized block can easily be transformed to a synchronized method. Transforming a synchronized method is as easy as moving the synchronized keyword one line down and add a curly brace at the end of the method to close the newly created synchronized block.

---

**Listing 2.2: Method synchronization**

```
public class Example{
```
private object sharedObject

public synchronized void doSomething(){
    if(sharedObject == null)
        createSharedObject();
    else
        performActionOnSharedObject()
}

The code-listing below shows a more fine-grained synchronization approach with the Double-Checked Locking [17] design pattern to remove unnecessary monitor contention. Note that DCL as implemented below is broken [16].

Listing 2.3: Statement synchronization with DCL

public class Example{
    private object sharedObject

    public void doSomething(){
        if(sharedObject == null)
            synchronized{
                if(sharedObject == null)
                    createSharedObject();
                else
                    performActionOnSharedObject
            }
        else
            performActionOnSharedObject()
    }
}

The concept behind synchronization is simple and easy to understand. Using more fine-grained locking increases scalability but also creates a more complex system where the programmer has to choose between different monitors.

2.2.2 ReadWriteLock

One of the latest additions to the Java API is the java.util.concurrent package. It offers several high-level (not part of the language but of api) locking mechanisms. ReadWriteLock is one of these mechanisms and makes a difference between read and write-locks. The idea behind this concept is that read-only access doesn’t change the state and read actions can take place in parallel (a counting semaphore). A write lock blocks other writes and readers until the lock is released (a binary semaphore). It also offers the concept of Reentrancy where re-acquiring a lock is possible (in recursion) and two algorithms for determining the next thread that can acquire the lock. [14]. According to Brian Goetz [2] these new higher level locking mechanisms should be used when features such as timed lock waits, interruptible lock waits, non-block-structured locks, multiple condition variables, or lock polling are needed. Besides these advantages, the java.util.concurrent package also solves the following limitations of synchronized [1]:

- No way to back off from an attempt to acquire a lock that is already held, or to give up after waiting for a specified period of time, or to cancel a lock attempt after an interrupt.
- No access control for synchronization.
- Synchronization is done within methods and blocks, thus limiting use to strict block-structured locking.
The downside of this type of locking is the complexity and the fact that releasing the lock does not take place automatically. Special care must also be taken when unchecked exceptions are thrown. An implementation of a ReadWriteLock mechanism in the example above is shown in code listing 2.4.

Listing 2.4: ReadWriteLock with valid DCL

```java
class Example{
    private final Lock lock = new ReentrantLock();
    private object sharedObject

    public void doSomething(){
        try{
            lock.readLock().lock();
            if(sharedObject != null)
                performActionOnSharedObject();
        }
        finally{
            lock.readLock().unlock();
        }

        //take a writelock
        try{
            lock.writeLock().lock();

            if(sharedObject == null)
                createSharedObject();
            else
                performActionOnSharedObject();
        }
        finally{
            lock.writeLock().unlock();
        }
    }
}
```

2.2.3 Transactional Memory

One of the latest developments in the field of concurrent programming is called (Software) Transactional Memory. The critical section in which shared state is accessed is wrapped into a transaction. Such a transaction can be started, aborted, committed or rolledback. A rollback only takes place when the transaction is aborted or when the commit action fails. A proper implementation of such transactional system should have a set of properties that guarantee reliable processing of its transactions [11]. The set of properties is also known under its acronym ACID (Atomic, Consistent, Isolated and Durability).

Transactions within the concept of Transactional Memory apply these properties on the fields and keeps the memory in an in consistent state. To achieve this, transactions are executed in isolation (changes stay within transaction until commits phase) and are committed atomically. In the commit phase a lock on the ’real’ fields is acquired and the mechanism checks whether the state has changed or not. In the latter case the commit succeeds and the state of the field(s) is changed, finally the lock is released. When a commit fails it rollbacks (semantics of rollback depend on implementation) and the transaction retries [7] [15] [5] [12].

This type of locking can be categorized as optimistic locking: a transaction keeps on running until commit phase where it obtains a lock and checks if the state has changed during execution. Although this concept of Transaction is not completely new as it is common within the context
of database, it has never been used as an abstraction to simplify parallel programming. Besides theoretical concepts, several implementations have been developed (DSTM, LSA-STM, AtomJava) for several programming languages such as Csharp Haskell and Java. These implementations vary in how the paradigm is added to the language but also how the transactions are administrated and the locking mechanism that is used within the commit phase.

**Deuce** Deuce is intended as a platform for developing scalable concurrent applications and as a research tool for designing new STM algorithms [15]. It is a noninvasive implementation of the TM paradigm and requires no language extensions or intrusive API's, and it does not impose any memory footprint or GC overhead. Being a high level framework, Deuce does not guarantee that concurrent access to a shared memory location both inside and outside a transactions are consistent. Deuce provides several benefits over existing Java STM frameworks:

- Avoids any changes or additions to the JVM.
- Does not require language extensions or intrusive APIs.
- Does not impose any memory footprint or GC overhead.
- Dynamically instruments classes at load time.
- Uses an original field-based locking strategy to improve concurrency.
- Provides a simple internal API allowing different STMs algorithms to be plugged and test.

The framework is delivered with two different algorithms who are responsible for managing the transactions within the system. These algorithms are responsible for processing results of committed transactions such as updating memory values. Such an action should also be performed atomically. Depending on the implementation a different mechanism such as an optimized form of locking or an atomic compare-and-set is used.

The following code-snippet demonstrates the easy use of Deuce with the single addition of the Java annotation 'atomic':

```java
public class Example{
    private object sharedObject
    @atomic
    public void doSomething(){
        if(sharedObject == null)
            createSharedObject();
        else
            performActionOnSharedObject();
    }
}
```

The algorithm used for managing the transactions is an implementation of the TL2 algorithm. It uses the notion of revokable versioned locks, with each object field within a critical section being mapped to a single lock. Each lock has a version that corresponds to the commit timestamp of the transaction that updated the field. When reading data, a transaction checks that the associated timestamp is valid, i.e., not more recent than the time the transaction has started and keeps track of the accessed location in its read set. When writing, the transaction buffers the update in write-set. At commit time, the transactions acquires (upon an atomic compare-and-set operation) the lock protecting all write-fields, verifies that all entries in the read-set are still valid, acquires a unique commit timestamp, writes the modified fields to memory and releases the locks. If locks cannot be acquired or validation of read-fields fails, the transaction aborts, discarding buffered updates. [15]
Part II

Case analysis: GX WebManager
With the knowledge that multi-core processors are the future and that writing highly scalable applications is not straight forward we now turn to GX WebManager. Within this part GX WebManager, specifically the part responsible for delivering webpage contents (Content Delivery). The first chapter will discuss the architecture of GX WebManager to give an approximate idea of how the system works. Within this context the concurrent characteristics will be analyzed such that it becomes clear which specific classes become a bottleneck at runtime. Note that it is not the goal of this thesis to improve the scalability of GX WebManager but to investigate the latest developments within shared state protection mechanisms.

This analysis is then used to start the actual measuring of the cache component within GX WebManager. This component is high concurrent, fulfills an important role in the process of content delivery and is of the right size for investigation. The construction of the test setup, the results, analysis and conclusions are part of the third chapter.
Chapter 3

Architecture of GX WebManager

GX WebManager can roughly be divided in an editorial environment (where editors work on content) and content delivery (that delivers content to the end-user). In next sections the architecture and concurrent characteristics of GX WebManager will be discussed with an extra focus on the Content Delivery part as this component plays an important role in the user experience.

GX WebManager is not a stand-alone Java application but runs within a so called web container. A web container is responsible for managing the execution of a JSP page and servlet component\[13\]. When a visitor types in http://www.gxsoftware.com, the browser sends a request that is received by the web container and is redirected to the corresponding web application. Depending on the architecture of this web application the entry point will be a Java Servlet or JSP page. Static resources such as an image or a static HTML page are handled by a separate Apache webserver for performance reasons.

Figure 3.1: Architectural overview of GX WebManager content delivery

Figure 3.1 shows a complete overview of all components of the content delivery part of GX
WebManager. It shows a visitor request passing through a filters component (which can modify and extend the request with new information) before reaching the cache component. This cache component is responsible for returning the actual contents (HTML, XML) back to the requesting visitor (browser). The entry point of the Cache component is the ShowServlet. The sequence diagram shown in 3.2 shows which classes are used for retrieving the contents and how these contents are streamed back to the browser.

![Sequence diagram of the cache component](image)

Figure 3.2: Sequence diagram of the cache component

The ShowServlet needs to construct an instance of the ContentStreamer class and pass an instance of a Connection to the handle method. A Connection contains the actual raw contents of a web page. These contents are stored in memory or on disk (depending on configuration parameters). To obtain a Connection, the ShowServlet needs to acquire a manager for the requested URL, which is known as an URLManager. One of the most important and high concurrent methods of this class is the method for retrieving the requested connection. The process of returning this connection is sketched in figure 3.3 on the following page and is quite complicated due to several process optimizations and protection of shared state.

The path a cache-request follows depends on the state of the cached contents. When these contents have the OK-state the connection to the cached contents is returned almost immediately. In some situations the cached contents are outdated but still acceptable. In such a situation the old connection is returned and a process of updating the current contents is scheduled. When this process is completed the current contents are swapped with newer (also known as ‘live’) contents. In other scenarios the contents are not usable and outdated and a process of retrieving newer contents is started. The ‘connection’ object that is accessed multiple times within this process is shared amongst multiple threads and therefore needs to be protected. This protection also controls the process flow as only thread can start or schedule a page-update. The effects of the protection mechanism is discussed in the next chapter.
Figure 3.3: Sequence diagram of the cache component
Chapter 4

Analysis of concurrent properties of GX WebManager

The entry point for front-end requests at GX WebManager is the ShowServlet. Before a request reaches this entry-point is does not share any state or resources with other concurrent requests. The web server component of the J2EE Server listens at TCP ports (mostly 80 and 8080 ) and forwards such a request to the entry point of the corresponding web application. Internally the J2EE Server uses a ThreadPool to pool requests that cannot directly be handled when there are not threads available. Limitations at this level are easily solved by adding more threads to the pool and adding more resources such as memory and processors.

The first shared resource that a page request encounters is the ShowServlet. Every page request is handled by the same block of code within this ShowServlet and therefore the J2EE Servlet specification requires that code needs to be thread-safe \cite{3}. The ShowServlet meets these requirements by protecting access to public fields within the ShowServlet.

**Thread-safe** For a class to be thread-safe, it first must behave correctly in a single-threaded environment. If a class is correctly implemented, which is another way of saying that it conforms to its specification, no sequence of operations (reads or writes of public fields and calls to public methods) on objects of that class should be able to put the object into an invalid state, observe the object to be in an invalid state, or violate any of the class’s invariants, preconditions, or postconditions. Furthermore, for a class to be thread-safe, it must continue to behave correctly, in the sense described above, when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, without any additional synchronization on the part of the calling code. The effect is that operations on a thread-safe object will appear to all threads to occur in a fixed, globally consistent order.\cite{9}

An important component within GX WebManager is the Cache component, because it significantly reduces the response time of the visitor’s request by directly returning the contents from the cache instead of generating a fresh page for every request. It is very common such a page is requested by multiple visitors at once, for example the home page of website. This Cache component will be the next stop for a page request and will encounter many shared objects that expose state.

4.1 The Cache component

The main task of the Cache component is to return contents for a given URL. Preferably, these contents should be retrieved from the cache instead of being generated. Because all cacheable page requests pass through this component (and many pages are cacheable) it has become a performance and scalability critical point within the whole design of GX WebManager. It needs to return the cached contents quickly but also needs protect access to objects that expose shared
state. An example of such an object is the URLManager which manages the status of the cached page. One of the first steps within the cache component is to lookup (or create if not existing) an URLManager using the requested URL. The URLManager is then responsible for fetching the contents of the page from cache or from the generator (responsible for generating fresh contents). The next two sections discuss these two steps in detail.

### 4.1.1 URLMap

The first object that is accessed by multiple threads at the same time is a HashMap. A HashMap can be described as a list of key-value pairs. In this situation the key is represented by an URL and the value will be a reference to an URLManager. When WebManager is started for the first time, the map will be empty. Every time a page request is received, the URL of this request is lookup up in the map. If this URL is found then the page request will be further handled by the corresponding URLManager. When such a URL cannot be found within the map then a new URLManager is instantiated and added to the map with the URL as its key. There is a special mechanism based upon SoftReferences that removes items from the map when the heap becomes too full [19].

If the map was immutable it could be accessed safely by multiple threads at the same time but it’s functionality would be very limited. The state of the map can be changed in two situations: a URLManager can either be removed or added to the map. Not all changes made to the map (and thus altering the state) will be directly visible to all threads. When this behavior is combined with the fact that threads are executed in a non-deterministic way a race-condition can occur and the map can be in a totally different state than a thread expects its to be. For example when Thread T1 constructs a new URLManager and wants to add it to the map. The situation may occur that thread T2 also notices that no URLManager exists for the requested URL just after T1 has started but before T1 has completed. In this situation the URLManager of T1 will be overwritten by T2.

A locking strategy called a ReentrantReadWriteLock is implemented to prevent situations described above. The advantage of this locking strategy is that read access to a protected variable does not block other readers. When write access is needed, the lock blocks succeeding readers and waits for current readers to complete their operations. The lock is released once the writer completes operation on the protected variable. This locking strategy affects the throughput and may eventually become a bottleneck, but only when the amount of write access increases because only one thread at a time can pass through. When only read locks are needed and the state of the internal variable doesn’t change no bottlenecks will occur.

Within GX WebManager the amount of write locks depends on the frequency of removing or adding URLManagers to the map. An URLManager will be added when it does not exist within the map, and removed when the heap becomes full. Removal of an URLManager is not random, the instance with the lowest access frequency will be deleted. A heap that is large enough to contain all the pages of a website will not suffer from URLManagers being removed or added from the map and thus should not encounter any performance degration due to the ReantrantReadWriteLock strategy. However, when the number of stored URLManager for the various web-pages becomes too large or the heap space too small the map cannot store all URLManagers in memory. This results into more write locks being requested to add or remove URLManagers and thus lowering the throughput.

### 4.1.2 URL Manager

Every page request is routed to a specific URLManager. An instance of this class takes care of organizing the retrieval of cached contents. Depending on the structure of the website and the browsing behavior of its visitors, the load will be divided among several URLManagers. However, some URL’s are visited more often than others, such as the homepage of a website.

The object that shares state amongst multiple threads within an URLManager is a reference to the cached contents as an instance of the class CachedConnection and is requested by every page
request that enters an URLManager. The state of CachedConnection changes when the cached contents become outdated and needs to be refreshed. Because this may happen at any time during execution all blocks of code that accesses the CacheConnection (which happens often) need to take measures to prevent illegal behavior.

Figure 3.3 on page 17 shows that these blocks of code occur frequently within the whole URLManager class. In this case a protection mechanism known as synchronization is used to control access to shared state and also assures that only one cache-update is issued when the current contents are outdated or acceptable. Although the mechanism is able to block many threads, in practice no performance issues occur as the critical section is quite short. However, when looking at the latest CPU developments and achieving the most effective solution, synchronized provides less scalability than possible. Adding more threads and more CPU resources to improve the throughput does not work and a bottleneck will develop for high amounts of concurrent requests.
Chapter 5
Measuring concurrent performance characteristics of the cache component

The goal of measuring the Cache component is to investigate how the component performs in a multi-processor environment and how scalability increases with more concurrent requests. When the amount of concurrent requests grows, is it useful to add more processors? These results will show inside information and reveals what the bottleneck of the cache component will be.

To simulate a concurrent environment, the test setup is constructed such that multiple threads request a set of 10 URL’s in parallel, with each thread sequentially looping through the set of URL’s. These tests are performed using 1, 2 and 4 processor cores and are rerun 3 times to improve data quality. When the test is run the response times are recorded to show how the system as its whole performs. When needed, more measurements can be performed to gain a better understanding of the results and what this means for the concurrent performance characteristics.

5.1 Test setup

The test-setup should simulate a concurrent environment and be able to take measurements such as average response times, throughput and monitor contention. To ensure that measuring don’t disturb the behavior of the Cache component and to simulate a most realistic situation the software that generates load and performs measurements should be located on a separate machine. Another machine should fulfill the role as ‘server-machine’ running a Tomcat server on which a simplified version of GX WebManager runs.

GX WebManager is stripped down to only a ShowServlet (for receiving requests and sending data to requesting clients), an URLManager and URLMap. Some supporting classes are needed for compilation but will only provide the core functionality needed. The URLManager and URLMap classes are also simplified to provide the core functionality while keeping the shared-state protection mechanisms intact. The load-generating machine is responsible for creating and sending the HTTP Request messages to the system under test. JMeter is a java application that can put heavy load on a web-application server by constructing and sending HTTP Request messages in parallel. It gathers several statistics such as average response times, standard deviation and throughput. To gather quality data, no other non-system processes are running on the server-machine and the load-generation machine has enough CPU resources to completely utilize the server-machine processors. Bottlenecks occurring outside the application (besides CPU utilization) are not desirable.

The Java Virtual Machine running Tomcat is loaded with a special agent supplied by a profiling tool called Yourkit that can gather statistics of a running Java applications such as CPU usage, monitor contention, memory usage and other metrics. The load-generating machine uses this agent
and aggregates these values. Finally a tool called sar is used to gather CPU info. This tool is more low-level than the one provided by Yourkit but offers more precise data on CPU utilization.

5.2 Hypothesis of test results

Based upon the description given in the previous sections, this section will try to predict what the expected concurrent behavior will be and how this affects the throughput of the cache component. In the most optimal (and theoretical) scenario, bottlenecks do not occur and the throughput scales linear with the number of threads under the assumption that hardware resources will be unlimited.

Of course such scalability is in practice not realistic. Not only due to the fact that unlimited hardware resources are not available but also due to the overhead that is created at several levels of the whole system in which the component runs.

In context of the simplified version of GX WebManager, the Cache mechanism will be the first point where resource sharing occurs, and as discussed in the the previous sections the URLMap will be the first class variable that is protected by a protection mechanism. Depending on the amount of possible pages and the maximum size of the heap space more or less write accesses will take place and influence the throughput. In the current test setup, the heap size will be large enough because only one page will be kept in the URLMap. This page is already initialized in the warm-up runs and cache-misses should not happen in the real test runs. With these parameters all page requests acquire a read lock and will not block concurrent threads. There will be no significant contention at the URLMap variable.

When all page threads concurrently pass through the URLMap they enter the responsible URLManager. In this instance synchronization is applied often to protect the access to the cached contents. Because only one page request can be handled by the instance other threads will be blocked. The time spent within a synchronized block is of importance of the time spent by other threads waiting. When a synchronized block is locked for 2 units of time, n succeeding threads will have to wait a minimum of 2 units and a maximum of n * 2 units of time to pass through this block of code.

Within this test setup, the cached contents are always up to date and will not change state. This property makes the time spent within the lock quite short but due to the fact that only one thread can be handled at once the effect on the throughput should be significant. This effect will become stronger when the number of threads is increased and more threads will be waiting for a lock.

Once a page request returns with the cached contents it will not encounter any shared resources anymore. Writing these contents back to the requesting client takes up both CPU and IO resources. In the case of a single CPU, the need for these resources will collide with other incoming page requests. The addition of more CPUs will spread the load that is generated by the writing process amongst multiple CPU’s and result in higher throughput scores. The effect of CPU addition is expect to be limited by the maximum possible throughput of the bottleneck within an URLManager.

5.2.1 Results

The graphs below shows the average of running 10 test scenarios four times with a pause of one minute in between. Averages were calculated for one, two and processor cores over 4 runs x 10 test scenarios x 3 processor configurations.

Note that that the throughput is not calculated from the average time. The average time is the time elapsed from just before sending the request to just after the last response has been received. Throughput is calculated as request / unit of time. The unit time is calculated from the start of the first sample to the end of the last sample plus the smallest average time (the last timestamp represents the time at the beginning of the last request). Time spent assembling the request is taken into account when computing the throughput and not when computing the average. Throughput numbers are therefore only applicable to the client and not to the server.
5.2.2 Analysis of results

This section will perform an analysis on the results presented in the previous section. These results distinguishes three different phases within the 10 test scenarios that are run on 3 different processor configurations.

- **Low load** - Less than 10 Threads
- **Medium load** - 10 Threads until 30 Threads
- **High load** - More than 30 Threads

The response times in the low load phase stay around 6ms for all processor configurations. Surprisingly the average response time for 1 thread is overall higher than for the other thread amounts within this phase. The same observation can be made when looking at the throughput figure. The throughput steadily increases with 20 pages per second every time the load on GX WebManager is increased with one more thread.

In the second phase the response times start increasing when more concurrent threads are active. Also a difference between the increase of response times for different processor configurations can be observed. In case of a single processor core the response times increase at the fastest rate. When two processor cores are available the increase is significantly smaller and is even reduced to the half of the response times for a single processor core configuration. Doubling the processor
cores to 4 doesn’t have the same impact but still reduces the response times for 20 en 30 threads with 5ms. The throughput also receives a large boost and is almost tripled for 30 threads and a 4 processor cores configuration when compared to 5 threads. The trend observed in the response times is also visible in the throughput graph; higher response times for a single processor core lead to a lesser increase of throughput when compared with configurations for two or four processor cores. Although the response times for the two and four processor core configurations are more than the half of the single processor core this does not translate to a doubled throughput rate.

The response times trend of the middle phase continues even more explicitly in the last phase as the relative response time ratio between for single, two and four processor cores configurations is kept. At the end of the last phase the response times for a single processor core configuration is a bit more than the double of the two processor cores, which is on its turn 15% slower than the response times for four processor cores. When this relative ratio is translated to concrete numbers the difference between a single processor core and two or four processor cores becomes clearly visible.

5.2.3 Explaining the results

The response times presented and analyzed in the previous sections are an accumulation of smaller portions spent by different parts of the whole system. Once one part consumes a relatively large amount of time it can become a bottleneck. The functionality of GX WebManager was reduced in such a way that it did not demand many system resources like memory, disk and network IO. An important factor that is expected to contribute to higher response times is the locking strategy that occurs at several parts in the Cache component. By using a profiler the time spent by waiting for a lock could be measured. Furthermore it was noticed that the processor utilization was often at 100% for a single processor core and around 60% per processor core of four. These two metrics were added to the test setup to investigate the build-up of the measured response times.

![Figure 5.3: Time spent waiting for lock on URLManager monitor](image)

The figure shows that the monitor contention is not a bottleneck for the test scenario run on a configuration with a single processor core. Time spent waiting for a lock is negligible when compared with the total response time. However, when two or four processor cores are used and the amount of threads reaches a minimum of 20 (phase Medium load) the average time waited for a lock becomes significant. The three phases with each a different rate of response time increase as identified in previous section (low/medium/high load) can also be observed in the figure above. Now it is not the response time that increases but the amount of time spent trying to obtain a lock.

The second metric is the CPU utilization caused by the Apache Tomcat web container process in which GX WebManager is hosted that handles the page requests. The next figures shows the utilization of 1 and 4 processor cores as a function of time.
Within each of the 3 figures above, ten peaks can be distinguished, each of them separately an amount of time where the load is almost zero. These peaks exactly match the ten test runs that take place (1, 2, 3, 4, 5, 10, 20, 30, 60 and 90 threads) in a test scenario. The 6th peak is the processor utilization measured during the execution of the test run with 10 threads. The middle phase as described in previous sections starts with ten threads and shows an increase of response times. The graph shows that for 10 threads the processor utilization is 80%. For a given time period the processor utilization is measured by taking several snapshots of the processor’s state. In this scenario, 80% of these snapshots showed that the processor was busy working on the Apache Tomcat process. For the rest of the middle phase the utilization reaches 100%. The rate of increase of response time steepens just when the processor utilization closes to 100%.

For 4 processor cores it is clearly visible that enough processor resources are available (minimum of 40% not utilized) and that the CPU utilization levels at at 60% for 30 threads and more (High load phase). When such a test run is started, all four processors are utilized for approximately 60% in the first second. Within the upcoming seconds the cpu utilization quickly reduces to 40% for two of the four processors while the other two still are used 60% of the time.
5.2.4 Conclusion

Based upon the analysis within the two previous sections the results can be divided into three different phases (low, medium and high load) with different characteristics for the processor configurations. The response times are consistent around 6ms for these processor configurations within the first phase. In this situation no bottlenecks occur and the throughput increases at the expected rate. Adding CPU resources within this phase has no effect as not all existing resources are completely used.

In the medium- and high load phase the response times and throughput produce different results for the different processor configurations. In a single processor core setup and the first test scenario of 10 threads (medium load phase) the processor utilization increases from 80% to 100% for 30 concurrent threads. The effect is that the processor is completely utilized and not all threads can be executed directly. Due to the maximum processor utilization a queue of threads waiting to be executed develops and the processor load surpasses 1.00.

This behavior continues more strongly in the high load phase as more concurrent threads are used. For a single processor core configuration the processor load becomes the bottleneck when more than 10 concurrent threads are performing page requests. The addition of another processor core doubles the maximum processor load and removes the bottleneck. As a result the response times are halved and the throughput is increased with 48% when more than 10 concurrent threads are active.

The effect of again doubling the amount of available processor cores to four is less than from one to two processor cores with a 15% reduction of response times and an increase of 10% when compared with a two core processor configuration. Using four processor cores does not introduce a significant performance improvement because the processor load is not the bottleneck anymore. The bottleneck is now moved to obtaining a lock which takes 80% of the response time for more than 20 threads and two or more processor cores.
Part III

Comparison of ReadWriteLock vs Software Transactional Memory
Chapter 6

Introduction

Locking mechanisms such as synchronized greatly reduce the scalability of an application; every addition of CPU resources becomes more inefficient up to the point where it has no effect at all. With the paradigm shift from frequency scaling to a multi-core concept as context; which shared state protection mechanism provides the best scalability and how effective does it use the increasing amount of cores?

This part of my thesis will compare the ReentrantReadWriteLock locking mechanism with Deuce as a software implementation of the fairly new concept of Transactional Memory. The next section will formulate a more precise research question with the main research question as context. The construction of the test setup and it’s parameters will complete the introduction of this introduction chapter. The next chapter will discuss and analyze the results and conclude by answering the original research questions.

6.1 Research question

The cache component in GX WebManager is highly concurrent and response times should be stable when the amount of concurrent requests increases. The same applies for other components within GX WebManager. In an ideal world, the application should scale perfectly over multiple cores and keep response times constant when more and more visitors perform page requests. However, we live in a less perfect world and the scalability is heavily reduced by the presence of shared state and mechanisms used to protect it. Other factors such as the amount of concurrent threads, complexity of the critical section and the amount of writes also affect the scalability of these mechanisms.

One of the problems of programs written in Java (and all other imperative programming languages) is to provide scalability as the load increases while preserving the semantical meaning of the program. A mechanism that protects shared state to preserve the semantics inherently reduces scalability as only one thread at the time can alter the state of the program (see chapter 2.1 on page 7). Shared state occurs at several levels within an application; it is not restricted to primitive types as public fields within a class but can also extend to the domain model. For example a webpage consists of several smaller components such as a heading, body and footer. How these components are related and which components are used makes up part of the state of a webpage object

ReadWriteLock and Transactional Memory are two kinds of mechanisms that protect shared state; but how scalable are they? Transactional Memory looks very promising due to its innovative concept, but does it scale better for different parameters such as rate of concurrency, complexity of critical section and amount of writes?

With these statements made and the overall research question as context, the following 3 sub-questions will be answered within this part of my thesis:
Which shared-state protection mechanism provides the best scalability as more CPU resources are added?

Which shared-state protection mechanism should be used for a given set of circumstances such as number of writes, complexity and load?

Which shared-state protection mechanism provides the best usability from the programmers point of view?

6.2 TestSetup

The goal of this test setup is to investigate how the two locking mechanisms perform for a set of factors that are expected to influence the scalability. A very simple web application contains almost two identical datastructures and is accessed by multiple concurrent threads. The two datastructures are identical from a semantic point of view but differ in the implementation of the locking mechanism. Both datastructures also contain a computation that represents the complexity of statements within the critical section. Requests to the web application contain an extra set of parameters. The web application will use these parameters to determine which locking mechanism should be used and call the requested function with the rest of the parameters. More info on these parameters can be found in section 6.3.

This test-setup is the same as described in 5.1 on page 21 and runs on a machine with four quad-core processors. To gather measurements about Deuce’s transactions a small modification has been made to the algorithm used. Every transaction that is started, committed or rollbacked and every field accessed is stored in memory and written to disk at the end of the test. This data was aggregated and related with the specific test that was running.

6.3 Test parameters

The characteristics of a locking-mechanism determine how scalable the protected shared state will be. These characteristics (and thus the scalability) can be affected by different external factors. Within this section I will describe these factors and show how these will be used to construct hypotheses about the ReentrantReadWriteLock and Transactional Memory mechanisms. Some of these hypotheses will be quite simple and may be even trivial but can be used to explain the more difficult hypotheses. Finally a set of hypotheses regarding the differences between the two locking-mechanisms will be constructed that should give answer to the initial research question.

Processor cores & concurrent threads In a purely theoretical situation where applications do not contain shared state the addition of more processors or cores will cause an linear increase (when CPU resources were a bottleneck). Adding more concurrent threads to the system will cause a linear increase in throughput (until CPU resources become a bottleneck). These two factors form the basis for testing the scalability of the two locking mechanisms. These mechanisms are needed because a stateless program is quite limited in its functionality. Adding these mechanism will reduce the scalability as it is impossible for a state to be modified by several threads at once. Furthermore, without such protection the semantics of the program will change and unexpected results may occur. The type of locking mechanism is therefore the third factor.

Locking mechanisms Both types of lock mechanisms treat read and write actions differently or expose different behavior when such an action occurs. A ReentrantReadWriteLock uses a binary semaphore for write locks and a counting semaphore for read locks while a concurrent write action with Deuce can cause another transaction to abort and retry. When designing an application that should scale properly, an architect should prevent the situation where multiple concurrent threads are writing to the same field. Write-actions greatly reduce scalability. This makes the lower percentage writes of most interest and will make up the fourth factor.
Complexity  The last factor influencing the scalability is the complexity of the computation within the lock. To simulate a complex computation a function that calculates a prime number [18] with a very slim chance that the number will not be prime. The complexity can vary through the length in bits of the prime number. Adding a single bit results into two times as many possible numbers (of which some are prime). Thus an increase of complexity has an expected exponential effect. Within a read-only situation, or very low percentage of writes a more complex computation within the lock will not have a very significant effect; it will only consume more CPU resources (which is, from scalability point of view good). However, in combination with a higher write percentage the complexity will cause the lock take longer to be unlocked and causing more threads to be blocked. For transactional memory systems it also has a negative effect: as running the transaction takes longer, the chance that another thread changes the state increases. Another effect is that retrying an transaction becomes a more expensive operation.

The list below summarizes the factors introduced within the previous paragraphs.

1. Number of processor cores (#cores)
2. Number of concurrent threads (#threads)
3. Locktype (rwlock and deuce)
4. Write percentage (% writes)
5. Complexity of calculation

The list below shows which types of data are collected by the test setup. This data will give more insight how the various factors above affect the scalability and response times.

- Response times (actually latency, not sure if it takes constructing request header into account)
- CPU Utilization
- Throughput in requests per minute (scalability)
- Reads/Writes/transactions started/transactions rollbacked/contexts created (for Transactional Memory only)
Chapter 7

Test results & analysis

7.1 ReadWriteLock

As explained in section 2.2.2 on page 10 the ReadWriteLock mechanism distinguishes between read and write locks. This approach has the advantage that ‘readers’ do not block each other and is therefore expected to provide higher scalability. However, once a write-lock occurs all other locks should wait until the write-lock is released. The next subsections will describe how the ReadWriteLock mechanism performs in aspect of response times, CPU utilization and throughput (factors influencing the behavior of the locking mechanism).

7.1.1 Response times results

![Figure 7.1: ReadWriteLock - Complexity 400 - 150 threads : Effect of writes on average response times](image)

**Write percentage**  An increase of write-accesses results into more write-locks that have to be obtained. The effect of this increase of write-locks is clearly visible (see figure 7.1) by the linear increase of response times. Another effect is that the addition of CPU resources has a very limited effect; only write percentages up to 10% show lower response times. With higher write percentages there are many write-locks obtained and less threads can run concurrently. As a result, adding more CPU resources has very limited effect for higher write percentages.

**Concurrent threads**  The increase of response times is also observed as effect (see figure 7.2 on the next page) of the increase of concurrent threads. Up to 16 concurrent threads this increase
Figure 7.2: ReadWriteLock - 10% writes - Complexity 400 : Effect of concurrent threads on average response times

seems to be acceptable with an average that is a 3.5 fold of a single thread situation. Doubling the amount of concurrent threads to 30 results in response times that are approximately doubled. From 30 concurrent threads and higher, the response time doubles as the amount of concurrent threads doubles: the critical section shows zero scalability. Adding more CPU resources makes a difference as of 20 concurrents threads and delays the zero scalability point. For higher response times the additional CPU resources have a more significant effect.

Figure 7.3: ReadWriteLock - 10% writes - 150 threads : Effect of complexity on average response times

Computational complexity The effects of the computational complexity in the critical sections are displayed in figure 7.3. The exponential nature of the complexity computation (see paragraph on complexity in section 6.3) can also be observed. This exponential increase of complexity causes the response times to rise quickly. This increase can be reduced by adding more CPU resources that can help executing the many complex computations. For 10% writes the reduction scales quite well with the addition of processor cores; the reduction caused by doubling the amount of cores from 4 to 8 is approximately the same as the step from 8 to 16. However, note that this scalability heavily degrades when the amount of write-actions increase (see 7.2 on page 32)

Average response times summarized The results in previous paragraphs show that a increase in write percentage, number of concurrent threads increase the average response time of the
protected critical section. With respect to scalability only the 0%, 10% and 20% write percentages are of interest as higher percentages do not benefit from more CPU resources. These extra resources have an effect as of 16 concurrent threads and the most as of 40. These amounts also display an increase of the response time growth. Computational complexity benefits from more CPU resources, although this scalability is only significant for the higher complexity and lower (less than 20%) write percentages.
7.1.2 CPU Utilization results

**Figure 7.4:** ReadWriteLock - Complexity 400 - 150 threads: Effect of writes on CPU Utilization

**Write percentage** When a write-action takes place, the pessimistic approach (always take a lock) of ReadWriteLock allows only one thread at a time to work on a critical section. Thus an increase in write-locks (as a result of an increase of the write percentage) causes the CPU utilization to drop. In a situation with 100% writes all threads need to pass one by one through the critical section and only one processor core is used for execution. Additional processor cores does not result in higher utilization for write percentages more than 20% as the rate of parallelism becomes too low.

**Figure 7.5:** ReadWriteLock - 10% writes - Complexity 400: Effect of concurrent threads on CPU Utilization

**Concurrent threads** An increase in the number of concurrent threads causes an increase of CPU utilization as there are simply more threads that are passing instructions to the processors. However the number of threads that is actually running depends heavily on the number of writes (see previous paragraph). When the percentage of writes is below 20%, doubling the amount of processor cores from 4 to 8 is only effective for more than 8 concurrent threads. 16 processor
cores make a difference for more than 16 concurrent threads. These boundaries will lower with a decrease of write-percentages as less write-locks need to obtained and more threads can run in parallel.

Figure 7.6: ReadWriteLock - 10% writes - 150 threads : Effects of complexity on CPU Utilization

**Complexity**  When the computation within the critical section becomes more complex it will require more CPU cycles to complete execution. In a scenario with no more than 20% writes the ReadWriteLock mechanism can benefit from more CPU resources as complexity increases. This effect is shown in the corresponding graph with 10% writes and 150 concurrent threads.

**CPU utilization Summarized**  The addition of more CPU resources does not always result in effective usage. Only in a situation with a percentage writes lower then 20% the CPU utilization increases. Within such a situation an increase of complexity or concurrent threads will cause more CPU resources to be effectively used.
7.1.3 Throughput Results

Figure 7.7: ReadWriteLock - Complexity 400 - 150 threads : Effect of writes on throughput

**Write percentage**  The ReadWriteLock mechanism is highly scalable when only read-locks are obtained as the throughput increases at a linear rate (doubling concurrent threads doubles throughput). In this situation the amount of blocked threads and/or time spent in blocked status is negligible and all threads can work in parallel. Adding more CPU resource has a significant positive effect. However this effect reduces and disappears for a write percentage of 20% and higher.

Figure 7.8: ReadWriteLock - 150 Threads - 10% writes : Effect of concurrent threads on throughput
**Concurrent threads**  Increase of concurrent threads will increase the throughput as more work can be processed within the same time period. However the time period can be heavily influenced by the amount of work that can be done in parallel. Figure 7.8 on page 36 shows that up to 8 concurrent threads the throughput increases linear. Beyond this point the amount of processor cores create a (small but significant) bottleneck as the high complexity consumes most of the CPU’s capacity. The same bottleneck appears at 12 threads and is temporarily solved when 16 processor cores are available. For more than 20 threads the scalability stagnates and more concurrent threads do not improve the throughput.

![Figure 7.9: ReadWriteLock -10% writes- 150 threads : Effect of complexity on throughput](image)

**Complexity**  An increase of complexity will decrease the throughput (it takes more time to process) and the CPU will become a bottleneck even faster. Again this bottleneck will be resolved by adding more CPU resources.

**Throughput summarized**  The scalability of the ReadWriteLock mechanism looks fine up to approximately 8 concurrent threads. Beyond this point the throughput rises more slowly while the amount of concurrent threads increases and the scalability is reduced. Amount of writes and complexity are factors that influence the point where scalability becomes less as high write percentages and complexity greatly reduce the throughput. Reducing the complexity in context of figure 7.8 on the preceding page will delay the point where the scalability starts to flatten. Also the effect of additional CPU resources is affected by the write percentage and complexity factor. Increases in write percentages reduce the positive effect of additional resources as the rate of parallelism greatly decreases with the increase of write-locks. When complexity increases the critical sections can benefit from these extra cores and the throughput increases.

**7.1.4 ReadWriteLock Conclusion**

ReentrantReadWriteLock is at the moment the most sophisticated solution of protecting shared state that is available as part of the Java API. It extends the 'synchronized' keyword with the concept of read and write-locks. When one of the factors that are expected to increase the characteristics of the locking mechanisms changes the effect was found in the results. An increase of complexity, amount of concurrent threads all caused the response times to increase. It provides near-linear scalability a situation with 0% writes.
A read-only scenario is not a very realistic situation for a locking mechanism, otherwise the shared state could have been made immutable, therefore the effects of an increase of % writes are of interest. A 10% write percentage already displays a significant effect: the response times rise, throughput and CPU utilization decrease. For lower complexity there is some scalability left at 20% writes, but beyond this amount and for higher complexity the ReadWriteLock does not seem to be scale up as the amount of threads increases. Adding more CPU resources has the largest effect for lower write percentages in combination with higher complexity.

The concept of pessimistic locking where threads are blocked has a negative effect on the rate of scalability. Scalability can only increase when threads are able to run in parallel to execute more work in the same amount of time. The effectiveness of extra processor cores also depends on this rate of parallelism and secondly on the computational complexity within the critical section.
7.2 Deuce

Deuce is a software implementation of the concept of Transactional Memory. Besides better scalability, Deuce/TM has several advantages (see section 2.2.3 on page 11) over the current locking mechanisms. But what is left of Deuce’s scalability when the number of writes and complexity increase?

7.2.1 Response times Results

![Figure 7.10: Deuce - Complexity 400 - 150 threads : Effect of writes on average response times](image)

**Write percentage** The percentage of writes indirectly affects the response time due to the fact that Deuce has no notion of locks (at least from the programmers point of view) and therefore is non-blocking. When the write percentage increases (which is caused by an increase of calls to the ‘put-method’ on the simplified datastructure) more transactions are to be cancelled and retried. All these retries puts more load on the CPU resources and this causes the response times to increase. Doubling the amount of cores from 4 to 8 and 16 shows a constant effect over all write percentages. Average response times are almost equal for a 8-core and 10% writes situation and a 16-core 20% writes situation. When only 4 cores are available and 20% of all accesses want to write; doubling the amount of cores reduces the average response time by a factor of 0.4. Only when the amount of writes increases up to 60% a 8-core setup comes on par with 4 cores and 20% writes. Adding more CPU resources reduces the average response times, but this effect becomes smaller every time more cores are added.

**Concurrent threads** Figure 7.11 on the following page displays the effect of concurrent threads on the response times. More concurrent threads results in a higher chance on rollbacks (as more write actions will take place). Up to 12 concurrent threads response times rise at a low linear rate. This rate increases at 12 and 40 concurrent threads. Additional CPU resources will reduce this increase for 40 concurrent thread and higher. The rate of increase for the three different numbers of cores is approximately the same.

**Complexity** Also the increase of complexity results in higher response times. However, this increase is not totally caused by the computation but also due to the fact that the chance for
a changed state at commit phase increases and the amount of rollbacks increases. Adding more CPU resources results in faster execution of the transactions and as a side-effect, less transactions need to be retried (see section 7.2.4 on page 45) Additional CPU resources are more effective for higher complexity and become less effective as more cores are added.

**Average response times summarized** Increase in the number of writes, complexity and amount of concurrent threads all result in an increase of the average response time of a critical section. Additional CPU resources reduce the response time for all 3 factors, however the effect of more cores reduces with the amount of cores. Furthermore, the effect of these extra cores is the largest for higher complexity and amount of concurrent threads. The effect of an increase of writes is reduced over it’s complete range (from 0 to 100) by adding more CPU resources.
7.2.2 CPU Utilization Results

Figure 7.13: Deuce - Complexity 400 - 150 threads : Effect of writes on CPU utilization

**Write percentage** The CPU utilization is not affected significantly for the complete range of write percentages. In a situation with 0 % writes no transactions need to be retried and all threads can run in parallel. This causes a maximum use of the CPU resources. Doubling these resources also results in almost a doubled utilization. For higher write percentages the same phenomena occurs, even up to 100%. In all these scenarios utilization increases when more processor cores are added.

Figure 7.14: Deuce - 10% writes - Complexity 400 : Effect of concurrent threads on CPU utilization

**Concurrent threads** Up to 4 concurrent threads (and complexity 400 % and 10% writes, see 7.14) the CPU utilization increase linear for 4, 8 and 16 cores. Beyond this point the 4-core
scenario stops increasing and flattens. Apparently the CPU resources are becoming too limited. Using an 8-core or 16-core processor setup the utilization keeps increasing up to 90 concurrent threads. An increase in the amount of concurrent threads increase CPU utilization at a linear scale. Finally a clear difference can be observed between the 4, 8 and 16 cores, this result is approximately the same as the difference in amount of cores.

![Figure 7.15: Deuce - 150 threads - 10% writes : Effect of complexity on CPU utilization](image)

**Complexity** Higher complexity causes more CPU cycles to be consumed while executing a transaction. This causes transactions to take more time to complete and therefore increasing the chance of a changed state at commit phase which results in aborting and retrying a transaction. These retried transactions will run in parallel with other transactions and consume the same amount of CPU cycles again. The last and third step of this effect is that these retried transactions can delay the other running transactions due to the limited amount of CPU resources that are available. Adding more cores resolves this problem by increasing the amount of CPU resources. This addition becomes more effective for higher complexity as within this situation the complexity takes more CPU cycles to complete and that there are more transactions running in parallel. This effect is shown in figure 7.15 as the utilization doubles when the amount of processor cores doubles.

**Average CPU utilization summarized** Complexity and amount of concurrent threads are factors that increase the CPU utilization when they increase. When these increasing factors are combined the effect of increasing CPU utilization is becomes greater. In this situation the addition of CPU resources becomes useful and the utilization increase even more.
7.2.3 Throughput Results

![Graph showing throughput results](image)

**Figure 7.16: Deuce - Complexity 400 - 150 threads: Effect of writes on throughput**

**Write percentage** Let’s start from a simple and less common situation where the amount of writes is zero. All requests can be processed in parallel and finally the CPU resources will create a bottleneck. In this situation, additional processor cores will remove the bottleneck and the critical section will scale up nicely (overhead not taken into account). In a more common scenario, the amount of writes will be higher than zero and the throughput decreases due to more retried transactions. At 10% writes, the throughput is already half of the 0% situation. Increasing the amount of writes will reduce the throughput even more, but this effect becomes smaller as the write percentages become higher. Also, the effect of adding more CPU resources becomes less effective for higher write percentages. A constant decrease is observed for 20% writes and less when the amount of cores is doubled.

![Graph showing throughput results](image)

**Figure 7.17: Deuce - 10% writes - Complexity 400: Effect of concurrent threads on throughput**
Concurrent threads  When the amount of concurrent threads increases and when these threads can work in parallel the throughput also increases. Due to the non-blocking behavior of DeuUsing Deuce all threads can work in parallel. Some of these threads execute transactions that are being retried. The results in graph 7.17 on the previous page show that a 4-core processor provides enough resources to scale up to 8 concurrent threads. Beyond this point the response times increase too much and total throughput stops increasing. These resources are becoming limited due to the complexity of the computation (see paragraph below) and percentage of writes (see paragraph above). Doubling the amount of cores to 8 results in an overall increase of 50%. Another doubling results in an 30% increase of the throughput. The extra CPU resources can execute more transactions at the same time and can decrease the effect of retried transactions. This effect is the major cause of decreasing throughput (and also increase response times and longer total execution) as more retrying transactions also consume CPU resources.

![Figure 7.18: Deuce - 150 threads - 10% writes : Effect of complexity on throughput](image)

Complexity  Increasing computational complexity will cause the total amount of CPU cycles to increase (see previous section on effect of complexity on CPU utilization). As a result the time to execute a transaction will increase and the throughput will decrease. Adding more cores becomes only effective for the higher range of complexity as low complexity does not utilize the total computational complexity.

Throughput summarized  More concurrent threads that run in parallel increase the throughput. However increasing write percentages and complexity reduce this effect by increasing transaction execution time. As a side-effect more transactions will be retried as the chance of changed state increases when transactions take longer to execute. Adding more CPU resources solves this bottleneck by allowing more threads to run concurrently and by reducing the effect of retrying transactions. For lower write percentages the scalability works quite well as the throughput increases with 50% when the amount of cores is doubled. For higher write percentages this effect reduces but still keeps some scalability.
7.2.4 Transaction results

Transactions are a new and important part of the Transactional Memory concept. This section will show the effects of complexity, amount of concurrent threads and percentage of writes on the amount transactions that are started and roll backed. This data is divided into two types of transactions: put and get. The transaction type refers to a critical section within the data structure that exposes shared state. A 'get'-transaction only reads the shared state while a 'put'-transaction reads and updates the value of the shared state.

![Deuce Complexity 400 - 150 threads: Effect of writes on transactions](image)

**Write percentage**  In a 0% write situation only get transactions are started and no rollbacks occur. The shared state is not modified and all transactions can commit without problems. Increasing the write percentage shows an increase of both put and get transactions that are started. The increase of put transactions is a direct result of an increase of write actions within the critical section and also due to the transactions that are roll backed. The amount of get-transactions that is started is the same for 10% and 60% writes. In the latter scenario, 50% of the started transactions are retried. Finally there are so few get transactions started that even retrying these cannot stop the decrease for higher write percentages.

Adding more CPU resources result into even higher amounts of transactions that are started, especially for higher write percentages.

**Concurrent threads**  The increase of concurrent threads has a two-folded effect on the amount of started transactions. The first effect is very trivial: every thread accessing the critical section will construct a transaction: increases in the amount of threads directly translates to more started transactions. When more transactions are running in parallel the average time until completion will increase as there are less resources per transaction. The same effect as described in the previous paragraph on write percentage will take occur: the chance of changed state increases as execution time increases and as a result more transactions will be retried. Additional CPU resources result into even higher amounts of retried transactions. Apparently the reduction in execution time is overruled by the increase of threads running concurrently; more threads running concurrent results into a higher chance of changed state.

**Complexity**  The effect of lengthening transactions is also clearly visible when the computational complexity changes. Higher complexity results into longer execution time of transactions, which on it’s turn increase the chance of changed state and retried transactions. Interestingly, the additionary CPU resources even increases the amount of started transactions.
7.2.5 Deuce Conclusion

Deuce is an software implementation of the Transactional Memory concept and uses an optimistic locking solution for protecting critical sections. In some sense the mechanism is also non-blocking.
as a transaction will be retried when the optimistic lock fails. It should be noted that the current
test setup is actually the worst scenario possible as all threads access the same fields within a
single critical section.

The effects of the optimistic locking strategy are clearly visible in the CPU utilization and data
regarding the transactions. An increase of the complexity, amount of concurrent threads causes
an increase in retried transactions which on their turn use more CPU resources. The phenomena
of retried transactions also increases the response times and reduce the throughput. The circle
is made complete as longer running transactions or more concurrent transactions result into a
higher chance of rollbacks as the started state will be changed. These metrics can be improved
by adding more cores to deal with the increasing CPU utilization. The interesting question then
arises: retrying transactions consumes CPU utilization: is the improvement of throughput and
average response times worth the almost total consumption of CPU resources?
7.3 **ReadWriteLock vs Deuce**

This section will show and discuss the differences in performance characteristics between the ReadWriteLock and Deuce shared-state protection mechanism. In its essence, Deuce is an optimistic locking mechanism which retries a transaction when it failed to commit (the pessimistic scenario). Its performance and scalability rely heavily on the number of aborted transactions. Research has shown that Deuce outperforms more common locking mechanisms both on performance as on scalability [15] in certain situations. But what happens when the situations gets a bit more pessimistic (higher percentage writes and longer locks) or more concurrent threads (strengthens the effect of retried transactions) reach the same critical section. It all comes down to ReadWriteLock having monitor/lock contention versus Deuce having transaction rollbacks.

In contrast with the previous two sections, the comparison will have a fixed complexity and only focus on the situation with 0%, 10% and 20% writes. The effects of complexity are discussed in previous sections. Also in previous sections it was concluded that higher write percentages than 20% are not of high interest. This section will be concluded with the complete range of factors (included complexity and write percentages larger than 20%).

### 7.3.1 0% writes

A situation with 0% writes results in no blocked threads or retried transactions. Both mechanisms should provide near-linear scalability and adding more CPU resources should also have a very significant effect. Differences between the two mechanisms will display the amount of overhead that occurs while managing access to the critical section.

![Average response time 0% writes](image)

---

The graph in figure 7.22 shows that there is no significant difference in response times between Deuce and ReadWriteLock. Secondly, doubling the amount of CPU resources reduces the response times. Note that this addition only pays off when CPU resources are becoming limited. Within this test-setup the bottleneck for CPU resources develops around 20 threads. At 150 concurrent threads the response times with 8-cores are exactly twice the response times of 16 cores. Also CPU utilization and throughput show the same trend.

As expected both mechanisms display near-linear scalability within a 0% writes situation. At 20% CPU resources become a bottleneck (see figure 7.24 on the next page) and doubling the amount of processor cores to 16 shows an extended scalability. Eventually at 40 concurrent threads, which is exactly twice the number of threads at the previous bottleneck (with 8 cores), both mechanisms start to level out due to a bottleneck within the 16 CPU cores. This bottleneck
is visible in figure 7.24 as the utilization level stops increasing. Furthermore the throughput is exactly two times as high as the situation with half of the processor cores and fits perfectly in the expectations.

Figure 7.24 shows the CPU utilization for the two locktypes Deuce and ReadWriteLock for 4, 8 and 16 cores as a function of amount of concurrent threads. No very significant differences occur up to 20 concurrent threads. Beyond this point some large differences occur. In the scenario of 0% writes both Deuce and ReadWriteLock are in the same magnitude of CPU resource utilization with Deuce using significantly more. Also a clear gap divides the 8 and 16 processor cores situations at exactly the right multiple of two. From these results can be concluded that the administration of Deuce takes a bit more resources than with ReadWriteLock but that this does not affect the average response times or throughput.
7.3.2 10% and 20% writes

**Response times** In a more common scenario where writes occur along with reads the effect of the locking mechanisms become clearly visible. Not only the response times become higher, but more importantly a performance difference between the ReadWriteLock and Deuce locking mechanism develops. Also the amount of processor resources not only influences the average response times but also shows how well a locking mechanism utilizes the extra cores. For four processor cores and both write percentages, response times of Deuce are significantly higher than for the ReadWriteLock mechanism. When the CPU resources are doubled both mechanisms perform equal and with 16 cores Deuce performs better for 90 concurrent threads and higher at 10% writes. Within a 20% writes context the behavior described in the paragraph above is also

![Figure 7.25: Average Response times - 10% writes](image1.png)

![Figure 7.26: Average Response times - 20% writes](image2.png)
shown within figure 7.26 on the preceding page. The increase of response times is a bit higher due to the fact that more write locks are obtained or more transactions fail in the commit phase. Deuce performs better when compared with the 10% writes situation: with 4 cores it keeps on a par with ReadWritelock up to 60 concurrent threads (instead of 30) and starts performing better using 16 cores with a lower amount of concurrent threads.

Throughput Results in the previous section showed that with 4 cores Deuce had the highest average response times. This is also visible in the lowest throughput score starting from 8 threads. Response times for both ReadWriteLock and Deuce at 8 cores were approximately equal and

![Figure 7.27: Average Throughput - 10% writes](image)

a quick look at the graph above confirms this as the throughput is equal over all numbers of threads. However a closer look reveals that between 12 and 30 concurrent threads Deuce has a lower throughput than ReadWriteLock. As the percentage of writes (see figure 7.28) or numbers of processor cores increases this effect becomes stronger. Overall can be concluded that Deuce provides better scalability but only when enough CPU resources are available. Also when the write percentage increase Deuce scales better than ReadWriteLock as for higher concurrent threads an 8-core Deuce situation is on par with a 16-core ReadWriteLock situation. Besides scalability over

![Figure 7.28: Average Throughput - 20% writes](image)

concurrent threads, the graphs above also show how the locking mechanisms scale over cores; how is the throughput affected as the amount of cores is doubled? Looking at ReadWriteLock, the
step from 4 to 8 cores has no significant effect on the throughput. Both the 4 and 8 core situation increase at the same rate with the increase of concurrent threads. The step from 8 to 16 shows a stronger effect: throughput is significantly higher and the throughput increases faster (up to 40 concurrent threads). Although the scalability over concurrent threads using 4 cores is almost none, the step to 8 and 16 cores is much larger when compared to ReadWriteLock. The scenario with 20% writes shows the same characteristics more stronger and Deuce scales better over threads and cores than ReadWriteLock.

**CPU Utilization**  
An increase in the percentage writes to 10% affects the CPU utilization with 16 cores and Deuce as locking mechanism enormously: from 20 threads and beyond the utilization skyrockets and leaves all other configurations behind. The 8 cores setup with Deuce still uses more CPU resources than a 16-core ReadWriteLock and a 4-core Deuce equals with 8-core ReadWriteLock. Also note that ReadWriteLock does not use the additional processor cores effectively as the step from 4 to 8 has no significant effect and the step from 8 to 16 is small in comparison with Deuce. Interesting point here is the effectiveness of the added CPU resources. Results in previous paragraphs show that Deuce scales better over threads and cores and does so by using at least two times the amount of processor resources used by ReadWriteLock.

![Average CPU Utilization for 10% writes](image)  
Figure 7.29: Average CPU Utilization - 10% writes
7.3.3 Conclusions

Before analyzing the differences between Deuce and ReadWriteLock, let’s first look at the context of this thesis. It is clear that software designers and engineers need to adapt their software designs and implementations such that they can benefit from multi-core processors, just as they did with the frequency scaling. Designing and implementing while preserving the semantical meaning of an application is not easy due to the concept of shared state. This thesis focussed on two of these mechanisms (ReadWriteLock and Deuce) and has compared them using the following three sub-research questions.

1. Which shared-state protection mechanism provides the best usability from the programmers point of view?

2. Which shared-state protection mechanism provides the best scalability as more CPU resources are added?

3. Which shared-state protection mechanism should be used for a given set of circumstances such as amount of writes, complexity and load?

Usability  It’s not just the performance of a shared-state protection mechanism that decides which mechanism is the best: usability also plays an important role. As explained in section 2.2.2 on page 10 Deuce and ReadWriteLock both use a different concept of marking critical sections. Although the concepts are different, they do not differ in expression capability. Both mechanisms also apply different strategies when threads access such a critical section. Deuce has no notion (from the programmers point of view) of locks and is therefore non-blocking. With it’s capability to nest transactions it has a huge advantage from a designers point of view: it is composable.

This property of Deuce is very important because the concept of a deadlock (a notorious problem in software construction) is not applicable. Programmers and designers don’t need to worry about obtaining and releasing the right locks and instead can focus on the actual work. Interfaces can also be designed with a cleaner contract as only the implementation is aware of the shared state and the protection mechanism used.

Although Deuce has no notion of deadlocks, it has to deal with the concept of a livelock. Luckily this type of failure is detectable and can therefore be solved. A simple solution (and is implemented within Deuce) is a maximum number of transaction retries. More sophisticated solutions should use the bookkeeping done by the system that manages all the transactions and give priority to particular transactions to commit.

Scalability  When an application is completely written and tested, it’s performance in a production environment is of importance as it affects the user experience. Scalability is, besides general speed, an important factor that shows how the performance develops as the load increases. Shared-state that can be shared amongst multiple threads should be protected to preserve the program semantics. A very simple but unscalable solution is the concept of synchronized where only one thread at a time can access the shared state. Lot’s of investigation has taken place on this subject to increase the scalability.

The concept behind ReadWriteLock is based upon the notion of locks, which categorizes this solution as blocking. The effect of this concept is clearly visible when the write-percentage factor increases: the throughput decreases rapidly. Once a thread obtains a lock other threads trying to obtain the same lock are blocked and need to wait, this is the main factor that reduces scalability. Increasing the complexity reduces the throughput even more. Depending on the previous two factors addition of more CPU resources has no effect (for 100% writes) up to maximum effect (for 0% writes). CPU utilization or the effect of adding more CPU resources is controlled by the number of write-locks that are obtained.

Deuce uses a different approach that is non-blocking and uses the concept of transactions. Such a transaction is retried when the initial state has changed during execution. The number of retried transactions influences the scalability as total execution time increases. This effect is
visible as the write percentage, complexity and number of threads increase. Also, CPU utilization rises as one of these factors increases. Adding more processor cores has a positive effect in all situations, even for higher write percentages. Depending on the number of retried transactions this effect can become more positive.

**Which mechanism for which circumstance?** Although both mechanisms serve the same goal the difference in concept also results in different characteristics in aspect of scalability. For both mechanisms scalability will decrease when more write-accesses need to take place within the critical section. When enough CPU resources are available, Deuce demonstrates better scalability and can effectively use the extra CPU resources. However, when CPU resources are limited ReadWriteLock is more effective as the resources are used more efficient.

When the critical section becomes more complex the CPU resource capacity is of importance (and is affected by write percentage). A low capacity brings Deuce in a disadvantage as many concurrent transactions will consume all these resources. Higher write percentages or concurrent threads strengthen this effect. With many transactions running in parallel, execution will take longer (less resources per transaction) when compared to ReadWriteLock where one active thread has more CPU capacity to claim. Additional CPU resources are almost completely utilized by Deuce and eventually it surpasses ReadWriteLock in performance and scalability. Although ReadWriteLock also benefits from more processor cores, it is bounded by the grade and type (write or read) locking that takes place. Finally it should be noted that the current set-up is a worst case scenario for Deuce whereas for ReadWriteLock no advantages develop within other scenarios.

With the multi-core trend as context, (Software) Transactional Memory is a very promising solution as its non-blocking property allows these extra cores to be utilized (in contrast to ReadWriteLock). The non-blocking property also increases the rate of parallelism within a critical section and therefore these extra cores are really needed to gain a better performance than a locking-solution.

**Consequences for GX WebManager** As discussed before, GX WebManager is a high concurrent webapplication with high demands with respect to response times, these should also be guaranteed when the load increases. When designing or re-engineering components of the application special attention is required to determine the characteristics of a critical section (reads/writes, complexity and an estimate of the number of concurrent threads). These parameters together with a concurrent flow analysis will help to determine which solution should be used now but can also be useful in the future when new solutions are become more mainstream. Some critical sections are protected using a synchronization mechanism, sometimes this mechanism also controls the process flow (such as starting a task for fetching fresh contents) besides protecting shared state. Although not the best solution, a better mechanism will not bring much improvement as the process itself becomes a bottleneck.

Within other situations the granularity of the critical sections becomes of importance. The java.util.concurrent.ConcurrentHashMap package locks at bucket level and provides better scalability when compared to a ReadWriteLock protected HashMap where the list of buckets is protected. When applied properly, shorter critical sections benefit both ReadWriteLock and Deuce. In situations where characteristics are poor, Deuce is a better choice but only when CPU resources can grow significantly. In other situations a cleverly crafted ReadWriteLock mechanism equals Deuce in performance but due to higher complexity is more expensive to maintain.

**Discussion** Deuce provides better scalability when enough CPU resources are available and also utilizes addition resources at an higher rate when compared with ReadWriteLock. This phenomena occurs due to the optimistic non-blocking concept that forms the basis of Deuce. Although this higher resource consumption fits Deuce better within the current processor development the question arises how effective the extra utilization is. Is a increase of 25% in throughput worth a quadruple of the CPU’s utilization? Where should the line be drawn?
Bibliography


